**Pseudocode Implementation**

## 1. File Input (Loading Courses into Data Structure)

### Vector Implementation (Milestone 1)

FUNCTION loadCourses(fileName)

OPEN file fileName

CREATE empty courseVector

WHILE file has lines

READ line

SPLIT line by commas INTO data[]

courseNumber ← data[0]

courseName ← data[1]

CREATE empty prerequisiteList

FOR i FROM 2 TO LENGTH(data) - 1

ADD data[i] TO prerequisiteList

END FOR

course ← (courseNumber, courseName, prerequisiteList)

APPEND course TO courseVector

END WHILE

RETURN courseVector

END FUNCTION

### ***Hash Table Implementation (Milestone 2)***

FUNCTION loadCourses(fileName)

OPEN file fileName

CREATE empty hashTable

WHILE file has lines

READ line

SPLIT line by commas into data[]

courseNumber ← data[0]

courseName ← data[1]

CREATE empty prerequisiteList

FOR i FROM 2 TO LENGTH(data) - 1

ADD data[i] TO prerequisiteList

END FOR

course ← (courseNumber, courseName, prerequisiteList)

INSERT course INTO hashTable USING courseNumber AS key

END WHILE

RETURN hashTable

END FUNCTION

### Binary Search Tree (BST) Implementation (Milestone 3)

CLASS Node

ATTRIBUTE course

ATTRIBUTE leftChild

ATTRIBUTE rightChild

FUNCTION Node(course)

SET this.course ← course

SET this.leftChild ← NULL

SET this.rightChild ← NULL

END FUNCTION

END CLASS

CLASS BST

ATTRIBUTE root

FUNCTION BST()

SET root ← NULL

END FUNCTION

FUNCTION insert(course)

IF root IS NULL

SET root ← new Node(course)

ELSE

CALL insertHelper(root, course)

END IF

END FUNCTION

FUNCTION insertHelper(node, course)

IF course.courseNumber < node.course.courseNumber

IF node.leftChild IS NULL

SET node.leftChild ← new Node(course)

ELSE

CALL insertHelper(node.leftChild, course)

END IF

ELSE

IF node.rightChild IS NULL

SET node.rightChild ← new Node(course)

ELSE

CALL insertHelper(node.rightChild, course)

END IF

END IF

END FUNCTION

END CLASS

## 2. Search and Print Course Information

### Vector Search

FUNCTION searchCourse(courseVector, courseNumber)

FOR EACH course IN courseVector

IF course.courseNumber EQUALS courseNumber

PRINT course.courseNumber, course.courseName

PRINT "Prerequisites:", course.prerequisiteList

RETURN

END FOR

PRINT "Course not found"

END FUNCTION

### Hash Table Search

FUNCTION searchCourse(hashTable, courseNumber)

IF courseNumber EXISTS IN hashTable

course ← hashTable[courseNumber]

PRINT "Course:", course.courseNumber, course.courseName

PRINT "Prerequisites:", course.prerequisiteList

ELSE

PRINT "Course not found"

END IF

END FUNCTION

### Binary Search Tree Search

FUNCTION searchCourse(BST root, courseNumber)

IF root IS NULL

PRINT "Course not found"

RETURN

END IF

IF root.course.courseNumber EQUALS courseNumber

PRINT "Course:", root.course.courseNumber, root.course.courseName

PRINT "Prerequisites:", root.course.prerequisiteList

ELSE IF courseNumber < root.course.courseNumber

CALL searchCourse(root.leftChild, courseNumber)

ELSE

CALL searchCourse(root.rightChild, courseNumber)

END IF

END FUNCTION

## 3. Menu Implementation and Course Listing

### Menu Options

FUNCTION displayMenu()

PRINT "1. Load Course Data"

PRINT "2. Display All Courses (Sorted Alphanumerically)"

PRINT "3. Display Course Information"

PRINT "9. Exit Program"

INPUT choice

SWITCH choice

CASE 1:

CALL loadCourses("course\_data.txt")

CASE 2:

CALL displayAllCourses()

CASE 3:

PRINT "Enter Course Number:"

INPUT courseNumber

CALL searchCourse(courseNumber)

CASE 9:

PRINT "Exiting Program..."

RETURN

DEFAULT:

PRINT "Invalid choice. Please try again."

END SWITCH

END FUNCTION

### Sorting and Displaying Courses:

### Vector Implementation

FUNCTION displayAllCourses()

SORT courseVector BY courseNumber

FOR EACH course IN courseVector

PRINT course.courseNumber, "-", course.courseName

END FOR

END FUNCTION

### Hash Table Implementation

FUNCTION displayAllCourses()

CREATE empty courseList

FOR EACH key IN hashTable

APPEND hashTable[key] TO courseList

END FOR

SORT courseList BY courseNumber

FOR EACH course IN courseList

PRINT course.courseNumber, "-", course.courseName

END FOR

END FUNCTION

### Binary Search Tree Implementation

FUNCTION inOrderTraversal(node)

IF node IS NOT NULL

CALL inOrderTraversal(node.leftChild)

PRINT node.course.courseNumber, "-", node.course.courseName

CALL inOrderTraversal(node.rightChild)

END IF

END FUNCTION

FUNCTION displayAllCourses()

CALL inOrderTraversal(root)

END FUNCTION

## 4. Runtime Analysis

The runtime complexity for each data structure:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data Structure** | **File Loading** | **Searching** | |  | | --- | |  |  |  | | --- | | **Sorting** | |
| Vector | O(n) | O(n) | |  | | --- | |  |   O(n log n) |
| Hash Table | O(n) | O(1) (avg) / O(n) (worst) | O(n) |
| BST | O(n log n) | |  | | --- | |  |  |  | | --- | | O(log n) (balanced) / O(n) (unbalanced) | | O(n) |

## 5. Advantages & Disadvantages

**Vector**

*Advantages:* Simple, direct indexing, dynamic resizing.

*Disadvantages:* Searching is slow (O(n)), inserting requires shifting.

**Hash Table**

*Advantages:* Fast lookups (O(1) average), minimal memory overhead.

*Disadvantages:* Potential collisions, worst case O(n) lookup.

**Binary Search Tree**

*Advantages:* Efficient searches (O(log n)), preserves order.

*Disadvantages:* Can degrade to O(n) if unbalanced.

## 6. Recommended Data Structure

After analyzing the three data structures, the best choice for storing and retrieving course data is the Hash Table.

*Justification*

-Fastest search time on average (O(1)), ensuring rapid lookups for course information.

-Efficient memory usage as data is stored with minimal overhead.

-Collisions are rare with a well chosen hashing function, keeping retrieval time optimal.

BST is a strong alternative butit requires balancing, and Vector is less efficient for large datasets. Therefore, Hash Table is the recommended structure for ABCU’s course management system.

## 7. Conclusion

Through pseudocode design and runtime analysis, this project evaluated Vector, Hash Table, and BST implementations for storing and retrieving course information. The Hash Table was determined to be the most efficient choice, balancing speed and reliability for academic course advising systems.